CSE 260 - Introduction to Parallel Computation

2-D Wave Equation
Suggested Project
Project Overview

Goal: program a simple parallel application in a variety of styles.
- learn different parallel languages
- measure performance on Sun E10000
- do computational science
- have fun

Proposed application: bang a square sheet of metal or drumhead, determine sounds produced

You can choose a different application, but check with me first.
Project steps

1. Write simple serial program. Oct 18
2. Improve serial program. Nov 1
3. Visualize and analyze output. Someday, perhaps
4. Write program in MPI. Nov 15
5. Write in OpenMP and/or Pthreads. Nov 22
6. Explore results. Various times along the way
2-D Wave Finite Difference Method

Let \( y_t(i,j) \) represent height of drumhead at location \((i,j)\) at time \(t\).

Square drumhead: \( i \) and \( j \) take on values in \(\{0, 1, ..., N\}\)

The formula:

\[
y_{t+1}(i,j) = 2y_t(i,j) - y_{t-1}(i,j) + c(y_{t}(i-1,j) - 2y_{t}(i,j) + y_{t}(i+1,j)) \\
+ c(y_{t}(i,j-1) - 2y_{t}(i,j) + y_{t}(i,j+1))
\]

should be combined

lets us compute all the \(y(i,j)\)'s for time \(t+1\), given values at \(t\) and \(t-1\).

We need:

“Initial values” for all the \(y\)'s at \(t = -1\) and \(t = 0\).

“Boundary values” for \(y(0,j), y(N,j), y(i,0)\) and \(y(i,N)\) for all \(t\).

Constant \(c\).
Step 1 - Simple serial program

• Program in C or Fortran.
  – Double precision (8-byte) floating point numbers

• Don’t use more than 32 $N^2$ Bytes of storage.
  – Otherwise, long runs will run out of storage.
  – Can use two or three 2-D arrays.

• Initial values (for $t=-1$ and $t=0$):
  $y(i,j)=1.0$ for $0<i<N/5$, $0<j<N/2$, $y(i,j)=0$ elsewhere.

• Boundary values:
  Four edges kept at 0.

• Constant $c = 0.1$
Step 1 - Simple serial program

• Write & debug program anywhere.

• Do timing runs on ultra (submit job from gaos).
  - You should get entire node to yourself.
  - Try several runs to see if times are consistent.

• Do timings for N = 32, 64, 128, ..., 1024.
  - Use optimization level 2.

• For each size, time program for 2 and 10 timesteps (in separate runs, or with call to gettimeofday).
  - Subtract to get “steady state” speed for 8 timesteps.
  - Make plot of “steady state cycles per point per timestep” versus N (problem size).
  - Note: ultra is 400 MHz, gaos is 336 MHz.
Selected Step 1 results

Cycles per iteration

Problem size

Fortran, fancy options
Step 2 - Tune the serial program

Goal - to get the one-processor version running at near peak speed.

- Inner loop has 5 floating point adds and 2 floating point multiplies.
  Actually, with extreme effort, can eliminate 1 add.

- UltraSPARC can execute 2 float ops per cycle
  But only if one is add and one is multiply!!

- 5 cycles/iteration is lower bound.
  6.9 was lowest in step 1, most had high teens or 20’s.

- <6 appears to be attainable for small problems
  Need to get several iterations going concurrently.
Step 2 Challenges

• Get inner loop to run well when data fits in cache
  - No more than 5 memory ops per point.
    • If inner loop is on “j”, shouldn’t load \(y(i,j)\) or \(y(i,j-1)\).
    • Can reduce loads still further if needed.
  - Does the compiler generate good address code?
    • Inner loop shouldn’t have any integer loads in it.
  - Does it have sufficient unrolling to overcome latency?

• Improve cache behavior for larger problem sizes
  - Does inner loop has stride 1 accesses?
  - Does compiler issues prefetch instructions?
    • Actually prefetching may not help.
  - How can you reduce the number of cache misses??
Project methodology

• malloc data (don’t use static assignment)

• Use gettimeofday, just around loop nest (e.g. don’t
time malloc)

• Also use unix “time” command (to ensure wallclock
is about equal to cpu time)

• There’s more information on the class website
(under “assignments”) about timing programs.

• You can use whatever compiler options you want.
  - I think you’ll learn more if you don’t just randomly try
    various option combinations

• No limit on memory usage.
Step 2 hand-in

- Due next Thursday (Nov 1)
- Run on ultra (not gaos)
- Tell what compiler and compiler options you used.
- Please provide program listing and assembly code of inner loop (e.g., via “-S”).
- Compare final values of untuned and tuned code: they should be identical!!
  - Conceivably, they'll be off in 15th digit.
  - Larger difference means there’s a bug in your program.
- As before, give cycles per point for problem sizes 32, 64, ..., 1024.
Improving cache behavior

Consider 1-d version

```c
p = 0; q = 1; /* p is t%2, q is (t-1)%2 */
for (t=2; t<T; t++){
    for(i=1; i<N; i++)
        x[p][i] = c*x[q][i] - x[p][i] + d*(x[q][i-1] + x[q][i+1]);
    p = 1-p; q = 1-q;
}
```

If N is huge, x and y won't fit in cache.

contents of x array outlined by rectangle

Iteration space

each square represents a stencil computation
Improving cache behavior

Consider 1-d version

```c
p = 0; q = 1; /* p is t%2, q is (t-1)%2 */
for (t=2; t<T; t++){
    for(i=1; i<N; i++)
        x[p][i] = c*x[q][i]-x[p][i]+d*(x[q][i-1]+x[q][i+1]);
    p = 1-p; q = 1-q;
}
```

Iteration space can be partitioned into “tiles”.

Execute all iterations is leftmost tile first, then next tile, ...

The amount of storage needed in cache is 2 time width of tile.
Improving cache behavior

Using parallelograms keeps storage use legal

```c
/* assume x[0] and x[1] are initialized */
/* “tile” with width W parallelograms */
for(ii=1; ii<N+T-3; ii+=W) {
    start_t = max(2,ii-N+3);
    p = start_t%2; q = 1-p;  /* p will be t%2 */
    for(t=start_t; t<min(T,ii+W+1); t++){
        for (i=max(1,ii-t+2); i<min(N,ii-t+2+W); i++)
            x[p][i] = c*x[q][i] - x[p][i]
                      + d*(x[q][i-1]+x[q][i+1]);
        p = 1-p; q = 1-q;
    }
}
```
Suggestion for 2-D wave equation

• Use tiles that are full width of matrix
  - to keep code from being too complicated

• Choose number of columns to easily fit in L2 cache.
  - For small problem sizes, can choose number of columns to fit in L1 cache.

• Interesting question: within a timestep in a tile, should you go row-wise or column-wise?
Step 3 of project – MPI version

• **Compile via:** `cc -fast -xarch=v8plus -lmpi ...
  - Other options are allowed too**

• **Submit:** `bsub -qhpc -m ultra -l -n 8 -W 0:1 a.out`
  - `-n 8` says “use 8 processors” (also use 1, 2, 4)
    • If you feel ambitious, you could more, but you need to use a batch queue
  - `-W 0:1` says “kick me off after 0 hours and 1 minute of CPU time”. Important particularly when program may be buggy!

• **Hand in** (Nov 15) program, running times and speedup relative to your tuned serial program, for 32x32, 256x256 and 1024x1024, for 1,2,4,8 processors, 100 timesteps.